

PVD¹: A Stable Implementation for Computing Voronoi Diagrams of Polygonal Pockets

Saurabh Sethia*, Martin Held**, and Joseph S. B. Mitchell***

State University of New York at Stony Brook, NY 11794, USA.

Abstract. Voronoi diagrams of pockets, i.e. polygons with holes, have a variety of important applications but are particularly challenging to compute robustly. We report on an implementation of a simple algorithm which does not rely on exact arithmetic to achieve robustness; rather, it achieves its robustness through carefully engineered handling of geometric predicates. Although we do not give theoretical guarantees for robustness or accuracy, the software has sustained extensive experimentation (on real and simulated data) and day-to-day usage on real-world data. The algorithm is shown experimentally to compare favorably in running time with prior methods.

1 Introduction

The Voronoi diagram (VD) is one of the most well studied structures in computational geometry, finding applications in a variety of fields. First introduced by the Russian mathematician Voronoi in his treatise [22] on the geometry of numbers, Voronoi diagrams have been studied and generalized in several directions over the last one hundred years. We refer the reader to [17] and [12] for detailed surveys of Voronoi diagrams and related structures, with their applications.

In this paper we consider the Voronoi diagram of a multiply connected polygon (a *pocket*), i.e. a polygon with holes, in which the *sites* are defined by the (reflex) vertices and (open) edges on the boundary of the polygon. The Voronoi diagram is defined as the locus of points that have two or more nearest sites. A Voronoi diagram divides the polygon into regions associated with the sites, such that all points in a region have a common nearest site.

Computing the Voronoi diagram of a polygon without using exact arithmetic, or at least “exact geometric computation” (EGC), is a very difficult task. The difficulty lies in the floating point round-off errors that are formidable due to the

¹ See <http://www.ams.sunysb.edu/~saurabh/pvd> for latest on pvd.

* saurabh@cs.sunysb.edu. Supported in part by grants from Bridgeport Machines and the National Science Foundation.

** held@ams.sunysb.edu. Partially supported by grants from Bridgeport Machines, the National Science Foundation (CCR-9732220) and Sun Microsystems.

*** jsbm@ams.sunysb.edu. Partially supported by grants from Bridgeport Machines, HRL Laboratories, NASA Ames, the National Science Foundation (CCR-9732220), Northrop-Grumman Corporation, Sandia National Labs, and Sun Microsystems.

high arithmetic degree¹ of our problem (estimated to be 40 [2]). These round-off errors can accumulate and lead to inconsistencies and failure.

In this paper we give an algorithm to compute the Voronoi diagram of pockets in the plane. Our algorithm accepts floating point input and performs only floating point operations. We have implemented the algorithm in C++ and, although we have not yet been able to show that it is *provably* robust, it is shown to be robust for all practical purposes. Even after testing on thousands of random and real-world inputs, we have not been able to break our code. The software has been named “pvd” (“pocket Voronoi diagram”).

Softwares that use exact arithmetic give exact results. However, there is a cost to pay for using exact arithmetic. We have designed our software to save this cost by not using exact arithmetic; as a consequence, the output diagram may not be exactly correct. There is a trade-off here. A comparative study of the two helps us to estimate the cost of using exact arithmetic. If, for a particular application efficiency is crucial and some amount of error is permissible, our software may be a good solution.

Computing offsets is one such application. An offset of a polygon is a polygon obtained by uniformly “shrinking” (or “growing”) the polygon. More formally, it is obtained from a Minkowski sum of the boundary of the polygon with a circular disk. Offsets of multiply connected polygons are used in numerically controlled (NC) contour milling. In NC milling, given a multiply connected polygon and a rotating tool, the goal is to “mill” (cover) the polygon using the tool. One of the approaches to do this is to compute successive offsets of the multiply connected polygon and then to “stitch” together these offsets (contours) in order to obtain a tool path. A thorough study of VD-based milling was done by Held [7]. He also studies various practical issues involving milling in general.

We have used pvd to compute offsets. In fact, our offset-finding routines form the core of the NC milling software “EZ-Mill” developed by Bridgeport Machines, where pvd has been in commercial use for the last two years.

1.1 Related Work

There are several algorithms for computing the Voronoi diagram of a simple polygon. The first $O(n \lg n)$ algorithm was given by Lee [13], who built on the earlier work of Preparata [19]. The problem of computing the VD of a set of disjoint polygons and circular objects was first studied by Drysdale in his Ph.D. thesis [4]. He achieved a sub-quadratic solution, which was subsequently improved by Lee and Drysdale to $O(n \lg^2 n)$ time [14]. Subsuming earlier work on computing VD of a set of disjoint line segments and circular arcs, Yap came up with a sophisticated worst-case optimal $O(n \lg n)$ algorithm [23]. For the special case of a simple polygon (without holes), a sophisticated linear time algorithm to compute the VD was given by Chin et.al. [3].

Several attempts have been made previously to implement the Voronoi diagram of a multiply connected or simple polygon or a set of line segments and

¹ An algorithm has degree d if its tests involve polynomials of degree $\leq d$ [15].

points. In the first systematic study, Held [7] implemented a divide-and-conquer algorithm similar to Lee's as well as an algorithm based on the wave propagation approach [8] outlined by Persson [18]. Although his implementation is very efficient, it is not robust and can crash. Another software (`plvor`) available is by Imai [11], who implemented an incremental algorithm for a disjoint set of line segments and points. His method guarantees "topological correctness". While his algorithm is worst-case cubic, its performance in practice is substantially better. His implementation gives no guarantee on the accuracy of the VD. Further, our tests of his code have led to unacceptably incorrect outputs in several cases. It is coded in Fortran and does not accept any floating point input but accepts only decimal inputs up to 6 decimal places. This is unacceptable for many applications. Another code (`avd`) is written by M. Seel based on exact arithmetic in LEDA [16]. This code gives the exact Voronoi diagram for any input. However, the use of exact arithmetic comes with a considerable penalty in the running time.

Very recently, the second author of this paper has developed another code (`vroni` [5, 6]), in the time since `pvd` was first developed and deployed in practice. `Vroni` uses techniques different from `pvd` to achieve its robustness and performance, though it also abides by the use of floating point arithmetic and carefully engineered robustness.

`Plvor`, `pvd`, and `vroni` do not guarantee accuracy of the VD. They only guarantee a form of "topological" correctness, which means that each site has a connected Voronoi region and the Voronoi regions of a segment and its end-points are adjacent. From a theoretical perspective this is not much of a guarantee. However, these softwares may perform well in practice and give practical solutions to real-world problems. The major strength of `avd` is that it does guarantee the exact computation of the correct VD. However, using software based on exact arithmetic may be prohibitive for some applications. A comparative study of software that uses exact arithmetic with software that uses only floating point computation helps us to understand the tradeoffs.

Also related to our work is the implementation by Sugihara and Iri for computing VD of a million points [21] and the implementation by Hoff et.al. that computes VD of segments and curves in two and three dimensions using graphics hardware [10]. Also of related interest is the work by Liotta et.al. on proximity queries in 2D using graphics hardware [15].

2 Simple Polygons

The Voronoi diagram of a simple polygon, with sites corresponding to the (open) boundary segments and the reflex vertices, has a very simple structure: it is a tree with some of the leaves "touching" each other at reflex vertices of the polygon. The leaf edges of the Voronoi tree are determined by consecutive sites on the boundary of the input polygon. If we ignore the geometry of the Voronoi diagram and only look at the underlying graph, we get what is called the *Voronoi graph*. The dual of this graph is the *Delaunay graph*, which, for non-degenerate inputs,

is a Delaunay triangulation. Degenerate inputs possibly lead to cycles of size larger than three; we handle such cases by arbitrarily triangulating such cycles, which is equivalent to allowing zero-length Voronoi edges in the Voronoi diagram (so that the degrees of all Voronoi vertices are three).

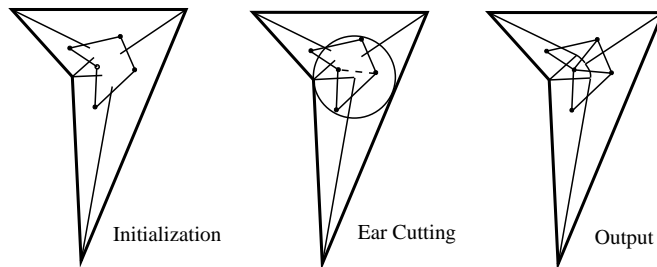


Fig. 1. Computing Voronoi Diagram and Delaunay Triangulation.

We know that there exists a Voronoi edge and hence a Delaunay edge between consecutive sites on the boundary of the input polygon. Thus, the outer cycle of the Delaunay triangulation is simply the cycle of consecutive sites of the input polygon. Our algorithm is based on simple “ear clipping”: It begins with this cycle and incrementally cuts “Delaunay ears” in order to obtain the final Delaunay triangulation. Here, an *ear* is a triangle two of whose sides are edges of the current cycle; a *Delaunay ear* is an ear such that the three sites involved determine an empty circle (a disk having no portion of the polygon boundary in its interior) touching them.

2.1 Delaunay ear cutting

Given a cycle of Delaunay edges, finding a Delaunay ear to cut is the central task in our algorithm. For a cycle of length n there are n possible ears, each of which can be tested for Delaunayhood by checking the empty circle property. Since any triangulation has at least two ears, we should, in theory, always be able to identify a Delaunay ear. Numerical inaccuracies, however, can result in a situation in which we do not find any Delaunay ear. In such cases we go into a “relaxed Delaunayhood” mode, described later. We do not need to check a linear number of ears every time we want to find a Delaunay ear, since every time we cut an ear, only the two neighboring candidate ears are affected.

2.2 Delaunay ear test

To check if an ear is Delaunay we first find the circle defined by the two edges of the cycle. The two Delaunay edges of the ear each correspond to a bisector between two sites. The first intersection of these bisectors gives the center of

a circle touching all three sites of the ear. The ear is Delaunay if this circle is empty.

We find the center and radius of this circle and check if it is empty by comparing its radius with the distance of its center with all other sites of the cycle. Thus, in the worst case, it can take linear time to check for Delaunayhood of an ear, leading to an overall worst-case quadratic time algorithm. In `pvd`, however, we utilize basic grid hashing so that we obtain, in practice, closer to linear time (or $O(n \log n)$ time) performance of our algorithm for most practical inputs. For polygons having a high degree of cocircularity (such as a circle approximated by numerous line segments), our algorithm does exhibit quadratic-time behavior. Future improvements planned for `pvd` will address this weakness.

2.3 Finding the ear circle

Finding the center of the circle corresponding to an ear is the only non-trivial task in our algorithm. Once we find the center of the circle, we take as the radius of the ear circle the minimum of the distance from this center to the three corresponding sites. (We have found that taking the minimum reduces the number of times we go into the relaxed Delaunayhood mode.) Also, we discard all circles that do not lie inside the bounding box of the input polygon as non-Delaunay. This is justified, since any Delaunay circle must lie inside the input polygon. This step not only improves the efficiency but also is vital for robustness. If we do not do this we can run into robustness problems with a circle whose center lies far away from the input polygon, resulting in the distances for the center to any site of the polygon appearing to be nearly the same, up to floating point precision.

In order to find the center of the ear circle, we need to intersect the two bisectors corresponding to the two Delaunay edges of the ear. We consider each Delaunay edge on the cycle to be oriented in the counter-clockwise direction of the cycle. A bisector corresponding to a Delaunay edge can be either a line bisector or a parabolic bisector, depending on its source and target sites. The source and target sites can be line segments or reflex vertices. A segment site may or may not have its endpoints as point sites.

We deal with several different types of ears in the code depending on the types of the three sites comprising the ear. However, there are essentially only three cases: ears with two line bisectors, ears with a line and a parabolic bisector, and ears with two parabolic bisectors. We describe our algorithm for each of the three cases. The primitives we use to implement these cases are described later.

For each of the cases we call the first site of the ear in counter-clockwise order p_1 or s_1 depending on whether it is a point or a segment. Similarly, we call the second and third sites p_2 or s_2 and p_3 or s_3 , respectively. We call the first bisector, i.e. the bisector between the first and second sites, b_1 and the second bisector, i.e. the bisector between the second and third sites, b_2 . Also, we consider the bisectors to be directed away from the boundary of the polygon. We refer to a point on a bisector as “before” or “after” another point on the same bisector according to the order along the directed bisector. We assume a

counter-clockwise orientation for the polygon. Thus the region of interest (the interior) of the pocket is always to the left of any segment.

There are certain conditions common to all three cases that must be satisfied. While these conditions are naturally satisfied for an exactly computed Delaunay graph, it is important to the robustness of our implementation that we specifically impose each of these conditions.

1. The first and third sites should be *eligible* to be adjacent in the Delaunay graph, where the eligibility criteria are:
 - Two point sites are always considered to be adjacent.
 - A point site can be adjacent to a segment site only if it lies on the left of the directed line containing the (directed) segment.
 - A segment site can be adjacent to another segment site if at least one of the ends of one segment is on the left of the directed line containing the other segment, and vice versa. See Figure 2.



Fig. 2. For two segments to be eligible to be adjacent each should have at least one endpoint on the left side of the other. (a). Segments are eligible to be adjacent; (b). Segments are ineligible to be adjacent.

2. The second bisector must cross the first bisector from its right to its left side.
3. If the first site is a segment, s_1 , then along b_1 , the center of the ear circle must lie before the point where b_1 intersects the perpendicular to s_1 at its source endpoint. See Figure 3.
4. If the third site is a segment, s_3 , then along b_2 , the center of the ear circle must lie before the point where b_2 intersects the perpendicular to s_3 at its target endpoint. See Figure 3.
5. The center of the ear circle must lie inside the bounding box of the input polygon.

Two line bisectors. This is a simple case in which the two bisectors intersect in one point or they do not intersect. If the necessary conditions above are satisfied, we get a valid ear circle.

One line and one parabolic bisector. In this case, the two bisectors can have zero, one or two intersection points. However, at most one intersection point can satisfy the first necessary condition. See Figure 4.

There is an additional condition that needs to be checked in this case. Note that if there are two intersection points, these two points divide each bisector

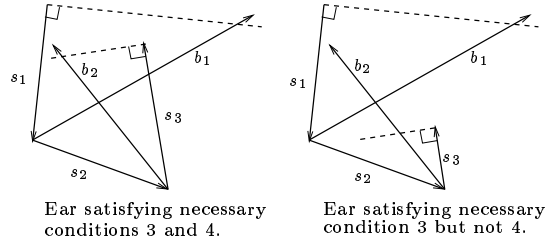


Fig. 3. Illustration of necessary conditions 3 and 4.

into three parts. We will call them the first, second and third part, respectively. Now consider the previous Delaunay circle for b_1 . It has to be empty; hence, its center must lie on the side of b_2 on which the second site lies. On that side, depending on the types of sites, either we would have the second part of b_1 or we would have the first and third part of b_1 . In the latter case we need to check if it is on the first or the third part, since, if it is on its third part, it means that we are moving away from the intersection points and therefore none of the intersection points are valid. An analogous check is also made for b_2 .

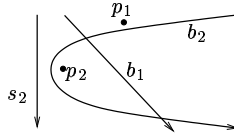


Fig. 4. One line and one parabolic bisector.

Two parabolic bisectors. In this case we compute the line bisector between the first and third site and find the intersection between this line bisector and the thinner of the two parabolic bisectors, thereby reducing this case to the case of one line and one parabolic bisector.

2.4 Geometric primitives

Sidedness primitive. A standard primitive is to decide whether a point C lies to the left, right or on the line defined by A and B . Most often, one does this by testing the sign of a determinant (the signed area of the parallelogram). We have found, however, that we are better off for robustness in using a pseudoangle based on slope, and then to decide sidedness based on the pseudoangles. In particular, we use pseudoangles based on partitioning 2π into 8 sectors of $\pi/4$ radians each, and mapping an angle $\theta \in (0, \pi/4)$ to a pseudoangle (slope) $m \in (0, 1)$. In this way, each angle θ gets mapped to a pseudoangle between 0 and 8.

Although less efficient than the standard signed area test, this test is much more robust for our purposes. The signed area can lead to trouble for cases when the area is very small. For example, suppose points A and B are extremely close to each other in comparison to distances to point C . In this case, irrespective of where point C is, the area is very small. Hence, for different values of C it can lead to conclusions that are not consistent with each other. Another example is when a slowly turning curve is finely approximated by line segments. In this case, any two consecutive segments appear to be collinear to a primitive based on area computation; then, a chain of such observations may ultimately lead to incorrect inferences.

Intersection of two lines. We find the intersection of two lines by solving the equations to find one of the coordinates and then using the equation of the first line to get the other coordinate. This ensures that even if we have numerical errors the intersection point lies some where on the first line. It is also crucial for robustness to choose which coordinate to compute first. We choose the one which avoids division by a small number.

Bisector of two lines. For our purposes, in most cases when we want to find the bisector between two lines, we already know one point on the bisector. Then, we first determine the slope of the bisector, which can always be determined robustly irrespective of the relative slopes of the two lines. This slope and the already known point determine the bisector. In cases in which we do not already know a point on the bisector, we calculate the bisector using elementary geometry. The equation of the bisector can be obtained from the equations of the two lines, using only additions, multiplications and square roots.

2.5 Relaxed Delaunayhood

As mentioned in Section 2.1, due to numerical imprecision our algorithm can get into a situation in which there are no Delaunay ears to cut while the triangulation is not yet complete. In such cases, our algorithm relaxes the condition for Delaunayhood, as follows. We now consider an ear to be “Delaunay” if it is *approximately* Delaunay: the circle corresponding to the ear is empty after its radius is reduced by multiplying it by $(1 - f)$ (while maintaining the same center point), where f is a fraction called the *shrink ratio*. Initially, f is taken to be the smallest floating point number such that $(1 - f) < 1$; it is then multiplied by 2 each time the algorithm cannot find a Delaunay ear to cut, until it succeeds, after which it is reset.

If the algorithm never goes into the relaxed Delaunayhood mode, it computes a Voronoi diagram that is accurate up to floating point round-off error. In the relaxed mode, the shrink ratio determines the measure of accuracy of the output. In fact, if f ever goes above 2^{-3} , we report an error. However, this has never happened in all the experiments we have conducted and all of the real-world use the algorithm has had to date. For most inputs the algorithm does not go into relaxed mode. For a few inputs, it enters the relaxed mode once, but it is rare to enter more than once.

3 Polygons with Holes

Our algorithm requires a Hamiltonian cycle of Delaunay edges at its initialization, after which it performs Delaunay ear clipping to determine the Delaunay triangulation. For a simple polygon, the Hamiltonian cycle is readily defined by the sequence of sites on the polygon, in order around its boundary. For a polygon with holes, the boundary consists of many such cycles; we need to link these cycles using some bridging Delaunay edges and then double them in order to obtain a single cycle. Thus we need to compute a “Delaunay spanning tree” (DST) of contours, which interconnects the contours into a tree using Delaunay edges.

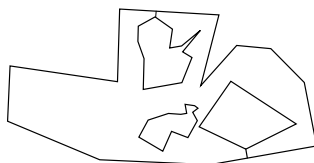


Fig. 5. Finding a Delaunay spanning tree (DST) for a pocket.

We find a DST of contours by incrementally finding Delaunay edges that bridge between connected components of contours. At any given stage, we attempt to find a bridge (Delaunay edge) linking an “island” contour to the set of contours that have already been connected by bridges to the outer boundary. We omit details from this abstract.

In the worst case, this bridging algorithm requires $O(kn^2)$ time. However, the use of simple bucketing makes it $O(kn)$ time in practice. For small k , this is efficient enough in practice; for large k , it becomes a bottleneck in our method, making it less effective for pockets having a large number of holes. Of course, in theory, we can implement more efficient methods (e.g., $O(n \log n)$); however, our objective was to keep the algorithm and its implementation very simple. We expect to improve the efficiency of the hole bridging in a future release of the code.

4 Efficiency Measures

Our algorithm to compute the VD of a simple polygon is worst-case quadratic, since we do $O(n)$ empty circle tests each of which may take $O(n)$ time. In contrast, there is a theoretically achievable linear time bound (which relies on linear-time triangulation, a result having no corresponding implementation). However, we achieve a running time much closer to linear (or $O(n \log n)$) time in practice by using simple grid hashing to make our empty circle test perform in constant time in practice. We use a grid of size $O(\sqrt{n}) \times O(\sqrt{n})$, resulting in $O(n)$ rectangular buckets. A site is associated with a bucket if it intersects the bucket.

Thus, a very long segment-site can lie in many buckets; a point-site will always lie in a single bucket. When performing an empty circle test, we examine only those sites that are associated with buckets that intersect the bounding box of the query circle.

5 Implementation

Pvd consists of about 8000 lines of C++ code and is available by request for research purposes, without fee. It is organized in the form of a library with a clean and simple interface to enable easy inclusion in client software. It has been compiled and tested on Unix using g++ and on Windows95 using Visual C++. We are in the process of testing it on other platforms.

6 Experimental Results

Pvd has sustained tests and real-world usage on hundreds of data sets obtained from boundaries of work-pieces for NC machining and stereolithography. The most important characteristic of pvd for our motivating applications is its robustness; it has always succeeded in computing the Voronoi diagram (or a close approximation thereto) for pockets. (The only exception to this are some cases in which the number of holes is extremely large, causing our current implementation of hole bridging to run out of memory.) The Voronoi diagrams that it computes are used every day by engineers and designers for computing offsets and automatic generation of tool paths.

Another important characteristic of pvd is its efficiency in practice. As part of our timing studies, we have conducted experiments on about 800 data sets created synthetically by various means, including the Random Polygon Generator (RPG) developed by Auer and Held [1]. Our experiments were carried out on a Sun Ultra 30, running Solaris 2.6 on a 296 MHz processor with 384 MB of main memory. All cpu times are given in milliseconds.

Table 1 shows the average time consumed by each software for random inputs of different sizes. Each entry is the average of cpu times (per segment) for 10 random inputs of the same size. The random inputs were generated by RPG. “n/a” means that the software did not finish in a reasonable amount of time. Pvd is seen to perform substantially faster than the prior methods (avd and plvor), while being somewhat inferior to the newest method of Held [6] that was just released (vroni [5]).

The main weakness we have observed in pvd is in its inefficiency in the cases of (1) numerous small segments that are nearly cocircular or that lie on a smooth curve (e.g., ellipse), since in this case our simple grid hashing is much less effective in empty circle queries, and pvd exhibits quadratic behavior; and (2) pockets having numerous holes, since our current implementation handles these inefficiently, using $O(kn)$ time and $O(k^2)$ space for k holes. (In particular, this can mean that we run out of memory for huge k .) The newly released vroni software does not seem to suffer from these weaknesses; see Held [6].

size	RPG			
	avd	plvor	pvd	vroni
64	50.55	1.359	0.164	0.109
128	75.26	0.922	0.160	0.107
256	119.8	0.657	0.162	0.108
512	209.6	0.565	0.158	0.115
1024	404.4	0.517	0.167	0.114
2048	758.9	0.511	0.167	0.117
4096	n/a	0.561	0.175	0.126
8192	n/a	0.643	0.185	0.136
16384	n/a	0.898	0.209	0.148
32768	n/a	1.293	0.234	0.156

Table 1. CPU time (per segment) for the different Voronoi codes applied to the “RPG” polygons. The “n/a” entries indicate that no result was obtained within a reasonable period of time.

7 Conclusion and Future Directions

We have reported on a system, *pvd*, which implements a simple and robust algorithm for computing the Voronoi diagram of polygonal pockets. The algorithm is carefully engineered to be robust in the face of floating point errors, without resorting to exact arithmetic. The overall algorithm is simple and all the crucial techniques that ensure robustness are restricted to a small number of primitives. The implementation is very practical from the efficiency point of view, except possibly in the case of a very large number of holes or “smooth” polygons. While we have not yet shown theoretical guarantees for our algorithm, our implementation has been extensively tested and in use for over two years now. We are also optimistic that one can in fact prove the algorithm to be robust and accurate, up to errors introduced by machine precision. More details on *pvd* would appear in [20].

Finally, we refer the reader to Held [5,9] for detailed information on the most recent system, *vroni*, which is based on an alternative incremental method of computing Voronoi diagrams robustly. *Vroni* has been shown to compare favorably with *pvd* in robustness, speed, and generality.

References

1. T. Auer and M. Held. Heuristics for the generation of random polygons. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 38–43, 1996.
2. C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *Proc. 2nd Annu. European Sympos. Algorithms*, vol. 855 of *LNCS*, pages 227–239. Springer-Verlag, 1994.
3. F. Chin, J. Snoeyink, and C.-A. Wang. Finding the medial axis of a simple polygon in linear time. In *Proc. 6th Annu. Internat. Sympos. Algorithms Comput.*, vol. 1004 of *LNCS*, pages 382–391. Springer-Verlag, 1995.

4. R. L. Drysdale, III. *Generalized Voronoi Diagrams and Geometric Searching*. Ph.D. thesis, Dept. Comp. Sc., Stanford Univ., CA, USA, 1979. Report STAN-CS-79-705.
5. M. Held. <http://www.cosy.sbg.ac.at/~held/projects/vroni/vroni.html>.
6. M. Held. VRONI: An Engineering Approach to the Reliable and Efficient Computation of Voronoi Diagrams of Points and Line Segments. To appear in *Computational Geometry: Theory and Applications*.
7. M. Held. *On the Computational Geometry of Pocket Machining*, vol. 500 of *LNCS* Springer-Verlag, June 1991.
8. M. Held. Voronoi diagrams and offset curves of curvilinear polygons. *Comput. Aided Design*, 30(4):287–300, Apr. 1998.
9. M. Held. Computing Voronoi diagrams of line segments reliably and efficiently. In *Proc. 12th Canad. Conf. on Comput. Geom.*, pages 115–118, Fredericton, NB, Canada, 2000.
10. K. E. H. III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. SIGGRAPH '99*, In *Comput. Graph.*, pages 277–285. ACM SIGGRAPH, Addison-Wesley, 1999.
11. T. Imai. A topology oriented algorithm for the Voronoi diagram of polygons. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 107–112. Carleton University Press, Ottawa, Canada, 1996.
12. R. Klein. *Concrete and Abstract Voronoi Diagrams*, vol. 400 of *LNCS* Springer-Verlag, 1989.
13. D. T. Lee. Medial axis transformation of a planar shape. *IEEE Trans. Pattern Anal. Mach. Intell.*, PAMI-4(4):363–369, 1982.
14. D. T. Lee and R. L. Drysdale, III. Generalization of Voronoi diagrams in the plane. *SIAM J. Comput.*, 10:73–87, 1981.
15. G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: An illustration of degree-driven algorithm design. *SIAM J. Comput.*, 28(3):864–889, 1998.
16. K. Mehlhorn and S. Näher. LEDA, a library of efficient data types and algorithms. Report A 04/89, Univ. Saarlandes, Saarbrücken, Germany, 1989.
17. A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Chichester, UK, 1992.
18. H. Persson. NC machining of arbitrarily shaped pockets. *Comput. Aided Design*, 10(3):169–174, May 1978.
19. F. P. Preparata. The medial axis of a simple polygon. In *Proc. 6th Internat. Sympos. Math. Found. Comput. Sci.*, vol. 53 of *LNCS*, pages 443–450. Springer-Verlag, 1977.
20. S. Sethia. PhD thesis, SUNY at Stony Brook, NY 11794, USA., May 2001.
21. K. Sugihara and M. Iri. Construction of the Voronoi diagram for ‘one million’ generators in single-precision arithmetic. *Proc. IEEE*, 80(9):1471–1484, Sept. 1992.
22. G. M. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. deuxième Mémoire: Recherches sur les paralléloèdres primitifs. *J. Reine Angew. Math.*, 134:198–287, 1908.
23. C. K. Yap. An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments. *Discrete Comput. Geom.*, 2:365–393, 1987.